**Git / Github**

Git est le système de gestion de versions (Version Control System ou VCS) le plus populaire. Il a été développé par Linus Torvalds, créateur du noyau Linux. Il offre un type de gestion de versions très différent, car il s'agit d'un système de gestion de versions distribué. Avec un système de gestion de versions distribué, il n'y a pas une base de code centralisée d'où tirer le code. Vous observerez cela plus en détail avec les dépôts locaux et distants (grâce à GitHub).

D'autres systèmes de gestion de versions, tels que SVN et CVS, reposent sur un système centralisé, ce qui signifie qu'une seule copie maîtresse du logiciel est utilisée. Ces systèmes nécessitent donc une connectivité réseau pour toute action au contraire de git qui va utiliser le réseau seulement lors d'une mise à jour entre le dépôt local et le dépôt distant.

L'intérêt de la gestion de versions (ou versioning) est de pouvoir revenir en arrière si besoin, et la capacité de voir qui a modifié quoi, a quel moment, ce qui apportera une meilleure gestion des participants du projet.

### Premier pas **Installation**

Git est installé par défaut dans de nombreuses distributions Linux. Il est notamment installé sur votre machine virtuelle DataScientest.

### **Initialisation**

Il faut avant tout créer un répertoire de travail. Une fois cette étape faite, il suffit de rentrer dans ce répertoire et d'utiliser la commande git init pour l'initialisation du dépôt en local.

mkdir repo

cd repo

git init

A la suite de cette commande, git crée un répertoire caché nommé .git . Pour rappel vous pouvez lister tous les éléments (y compris les éléments cachés) avec la commande ls -a. Vous pouvez rentrer dans le répertoire et faire un ls.

ls -a

cd .git

ls

C'est dans ce répertoire que vont être stockées toutes les données relatives à la création et modification d'éléments dans votre répertoire de travail. Ne prêtez pas forcément attention à ce qu'il y a dans ce répertoire, comprenez qu'il permet le bon fonctionnement de git.

**Ajouter des documents**

Lorsque l'on crée un fichier, pour que git prenne en compte l'ajout de ce fichier dans le dépôt il faut utiliser la commande git add nom\_du\_fichier.

La commande d'ajout peut permettre d'ajouter plusieurs documents en même temps, voire de prendre en compte toutes les modifications qui ont été faites dans le répertoire.

C'est possible en utilisant "git add .".

Il faut néanmoins savoir que c'est une pratique à éviter. En effet le but va être d'historiser au mieux vos ajouts/modifications. Pour qu'en cas de besoin il soit plus simple dans le futur de comprendre ce qui a été fait dans votre répertoire.

Plus tôt vous prendrez de bons réflexes avec git, mieux ce sera pour vos futures missions pour lesquelles vous utiliserez git.

**Faire un commit**

On a mentionné plus haut le fait d'historiser vos ajouts/modifications de documents. Pour accomplir cette tâche, à la suite d'un ajout (git add), on doit faire un commit c'est-à-dire enregistrer l'ajout ou la modification du document dans l'historique des modifications. On peut ajouter un message à nos commits pour garder une trace commentée de toutes les modifications. On va donc faire un commit avec un commentaire intéressant et pertinent pour les utilisateurs grâce à la commande git commit -m "le commentaire".

Exemples :

git commit -m "ajout du fichier nom\_du\_fichier"

git commit -m "modification de la fonction nom\_de\_la\_fonction"

git commit -m "ajout de documentation"

**Supprimer des documents**

Lorsque l'on souhaite supprimer un fichier, pour que git prenne en compte la suppression de ce fichier dans le dépôt il faut utiliser la commande git rm nom\_du\_fichier. Comme pour l'ajout ou la modification d'un document avec git add, il faudra également enregistrer la suppression avec un commit.

git rm "mon\_fichier.txt"

git commit -m "suppression du fichier mon\_fichier.txt"

**Connaître le statut de son dépôt local**

A tout moment, il est possible de connaître l'état du dépôt: grâce à la commande git status, git nous guide également sur les actions que l'on doit faire. Cette commande est extrêmement importante, il est conseillé pour bien prendre en main l'outil de toujours faire un git status avant et après chaque commande pour voir l'évolution de nos documents.

Lors de l'exécution de git status, si rien est à faire, on devrait avoir l'affichage suivant.

On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

On comprend donc les informations qui vont nous être fournies à savoir: Sur quelle branche sommes nous (notion qui sera vue plus tard); si il y a des fichiers à ajouter; si il y a des fichiers à commiter.

Exemples de commandes :

touch doc.txt

echo 'Mon premier fichier dans mon répertoire git' >> doc.txt

git status

git add doc.txt

git status

git commit -m "ajout du fichier de documentation"

git status

**Configuration**

Il est possible d'observer les variables de configuration de git à l'aide de la commande suivante.

git config --list

C'est ici que les informations concernant le fait qu'il y ait un dépôt distant ou non sont stockées. On peut également rajouter des informations concernant l'utilisateur. Par exemple, lors d'un commit il est important d'avoir les coordonnées (nom, email) de la personne qui a fait les modifications. Vous pouvez changer ces informations grâce à la commande suivante.

git config --global user.name "Daniel"

git config --global user.email  "[daniel@datascientest.com](mailto:daniel@datascientest.com)"

**Log**

Pouvoir observer les logs est essentiel pour un outil de gestion de versions. C'est une fonctionnalité très utilisée, notamment pour savoir quel utilisateur a fait quelle modification (et à quel moment). Cela permet de gérer les versions du répertoire de travail, d'avoir un vrai suivi des branches et de l'évolution des fichiers.

Faites une modification dans le fichier doc.txt puis faite l'ajout et le commit.

echo 'nouvelle ligne' >> doc.txt

git status

git add doc.txt

git status

git commit -m "ajout de ligne"

git status

Exécuter la commande git log et observer la sortie.

Tout d'abord on peut remarquer entre nos deux commits que l'auteur a changé car l'on a modifié la configuration. Il existe des options de git log pour avoir un affichage avec une sortie plus élégante ou des informations en plus :

* --oneline: Chaque commit est représenté sur une ligne.
* --graph: Représente la sortie sous la forme d'un graphe (utile lorsqu'il y a plusieurs branches).
* --name-status: Rajoute comme information le type d'action réalisée (A pour un ajout, M pour une modification)

Il en existe plein d'autres. Pour aller plus loin vous pouvez lire la documentation (commande man git-log) pour voir toutes les options disponibles, vous pouvez même combiner ces options! Voici des exemples avec les options combinées.

git log

git log --oneline

git log --oneline --graph

git log --oneline --graph --name-status #celle ci est un bon combo pour débuter avec git

**Différence entre deux commit**

Il se peut lors du développement d'une application qu'il y ait un incident en production ou dans un fichier que l'on a réalisé. Dans ce genre de situation, on veut rapidement voir les différences entre la version d'avant et d'après afin de voir les modifications, pour cela on utilise la commande git diff ou l'on passe en paramètre les numéros des deux commits a inspecter.

git diff numero\_de\_commit1 numero\_de\_commit2

Observer la différence entre vos deux commits. (des - et des + indique les ajouts et suppressions de ligne)

**Git Blame**

Une fonctionnalité intéressante de git est de pouvoir afficher les informations d'un document ligne par ligne, ainsi que l'auteur, la date du commit lié à au changement d'état du document. On peut pour cela utiliser git blame nom\_du\_fichier.

**Retour en arrière**

git revert numero\_du\_dernier\_commit

On peut aussi revenir en arrière de plusieurs commit en utilisant git revert pour annuler une plage de commit :

git revert HEAD~2..HEAD # revenir 2 commit en arrière

Cette commande créera deux nouveaux commits, qui annulent respectivement le dernier et l'avant dernier commit.

**git reset**

A la différence de git revert, qui crée des nouveaux commits pour annuler des commits passés, git reset supprime purement et simplement des commits de l'historique.

Ainsi, si l'historique des commits est le suivant : commit1 -- commit2 -- commit3 -- commit4, pour revenir à l'état du deuxième commit, on pourra utiliser la commande suivante:

git reset commit2 # revenir à l'état du commit commit2

Ces commandes auront pour effet de supprimer les commit commit3 et commit4. L'historique après cette opération sera donc commit1 -- commit2.

En comparaison, si l'on avait utilisé la commande git revert HEAD~2..HEAD, le nouvel historique serait commit1 -- commit2 -- commit3 -- commit4 -- commit5 -- commit6, les commits commit5 et commit6 ayant respectivement pour effet d'annuler les commit4 et commit3.

Si l'on souhaite annuler le précédent commit sans effacer les modifications faites en local lors de celui-ci, on utilisera l'argument --soft:

git reset --soft HEAD~1

# L'argument HEAD~1 précise que l'on souhaite revenir au commit précédent

# l'actuel étant (designé par HEAD)

Si l'on souhaite annuler le commit ainsi que les modifications faites durant celui-ci, on utilisera l'argument --hard:

git reset --hard HEAD~1

Utiliser git revert est considéré comme plus "propre", car on garde une trace de tous les commits, et de leurs suppressions.

## Branche

Lorsque vous créez une application, il est toujours souhaitable d'avoir une version stable de celle-ci, mais que se passe-t-il si vous souhaitez développer une nouvelle fonctionnalité ? Car vous ne voulez pas écraser la version stable, ou vous ne voulez pas avoir a faire des git revert sans arrêt en cas d'erreur. C'est là que l'intérêt des branches entre en jeu.

**Théorie**

**Your work**

Master

Someone else’s work

Par défaut une seule branche existe lors de l'initialisation du dépôt, la branche master (celle-ci a été renommée en branche main sur GitHub). La bonne pratique est de laisser la version de votre projet la plus stable sur celle-ci.

Lors du développement d'une nouvelle fonctionnalité vous serez amenés à créer une nouvelle branche. Le fait de créer une nouvelle branche va nous permettre de travailler avec les fichiers du répertoire courant dans la nouvelle branche. Ainsi vous pourrez développer votre nouvelle fonctionnalité sur la nouvelle branche, sans oublier les bonnes pratiques apprises jusqu'à présent (git add, status, commit...). Quand vous êtes satisfaits du travail réalisé sur la branche, il est possible de fusionner la nouvelle branche avec la branche master.

### **Pratique**

#### **Création de branche**

git branch nom-branche

Pour observer les branches qui sont à notre disposition on peut simplement faire un git branch.

**Changement de branche**

git checkout nom-branche

#### **Fusion de branche (merge)**

Enfin pour fusionner deux branches, par exemple si on veut apporter les changements de la branche branche-source à la branche master, il faudra d'abord se situer dans la branche master avec un git checkout puis utiliser la commande git merge :

git checkout master

git merge branche-source

Exercice

Créez une nouvelle branche modif-doc

Déplacez vous dans celle-ci

Faites des modifications dans un document (par exemple via nano)

Enregistrez ces modifications (add + commit)

Assurez vous que les modifications sont bien prises en comptes dans la branche (status + log)

Retournez dans la branche master et observez que les modifications de sont pas présentes

Fusionnez les deux branches et observez maintenant que les modifications se trouvent bien dans master

git branch modif-doc

git branch # l'astérisque indique la branche courante (c'est à dire sur laquelle on se situe)

git checkout modif-doc

git branch

nano doc.txt

git status

git add doc.txt

git status

git commit -m "modif doc.txt branche modif-doc"

git status

git log --oneline --graph --name-status

cat doc.txt

git checkout master

git branch

cat doc.txt

git merge modif-doc

cat doc.txt

git log --oneline --graph --name-status

En résumé, les branches permettent une meilleure gestion des versions des fichiers. Elles permettent également de travailler plus facilement à plusieurs (chaque personne peut travailler sur une branche distincte).

**Gestion des conflits**

Dans chaque situation où plusieurs personnes travaillent sur le même fichier, le travail finit par se chevaucher. Il arrive également que deux développeurs modifient la même ligne de code de deux manières différentes. Dans ces cas, Git ne peut pas dire quelle version est correcte, seulement un des développeurs peut décider.

Si cela se produit, un développeur verra l'erreur suivante pendant une fusion git :

Auto-merging [filename1]

CONFLICT (content): Merge conflict in [filename1]

Automatic merge failed; fix conflicts and then commit the result.

La résolution des conflits de fusion peut prendre une minute ou plusieurs jours (s'il y a beaucoup de fichiers à corriger). Il est recommandé, et c'est une bonne pratique de codage, de synchroniser votre code plusieurs fois par jour en faisant des commits, des pushs, et des merges régulièrement.

Exercice

Nous allons à présent voir un exemple de conflit et sa résolution:

Créez un nouveau dossier conflits

Entrez dans le dossier conflits

Initialisez le dépôt git

Créez un fichier merge.txt et insérer le texte première ligne

Faites l'ajout et le commit

mkdir conflits

cd conflits

git init .

echo "first line" > merge.txt

git add merge.txt

git commit -m "add merge.txt file"

Nous disposons à présent d'un nouveau dépôt avec une branche master et un fichier merge.txt avec du contenu. Nous allons maintenant créer une branche ou nous allons créer un conflit.

Créez une nouvelle branche update-merge-file

Modifiez le contenu du fichier merge.txt

Faites l'ajout et le commit

git checkout -b update-merge-file

echo "new version of first line" > merge.txt

git add merge.txt

git commit -am "updated merge.txt"

Retournez maintenant sur la branche master

Ajoutez une deuxième ligne dans le fichier merge.txt

Faite l'ajout et le commit

git checkout master

echo "second line" >> merge.txt

git add merge.txt

git commit -am "added second line in merge.txt"

Maintenant que nous avons deux versions différentes du même fichier a travers deux branches différentes, nous allons essayer de faire un merge entre ces deux branches.

Fusionnez update-merge-file dans master

git merge update-merge-file

Le message suivant devrait apparaître.

Auto-merging merge.txt

CONFLICT (content): Merge conflict in merge.txt

Automatic merge failed; fix conflicts and then commit the result.

Le message nous indique donc qu'il y a un conflit avec le fichier merge.txt. Un git status permet de comprendre que le fichier merge.txt a donc été modifié afin de régler le conflit. Nous allons voir à présent le contenu dans le fichier:

Nous pouvons voir certains ajouts dans le fichier:

* <<<<<<< HEAD
* =======
* >>>>>>> update-merge-file

Ces lignes permettent de montrer le conflit et comprendre la différence entre les deux branche: Tout le contenu entre la ligne <<<<<<< HEAD et ======= provient de la branche actuelle vers laquelle pointe la réf HEAD (en l'occurrence master). Et donc, tout le contenu entre ======= et >>>>>>> update-merge-file provient de la branche de merge (update-merge-file).

Afin de régler le problème, seul le développeur sait quelle version garder. Ainsi, la meilleur méthode pour résoudre celui-ci est de modifier modifier manuellement les fichiers en questions pour garder les versions souhaitées.

Modifiez le fichier Faites l'ajout et le commit

nano merge.txt

git add merge.txt

git commit -am "conflit résolu et merge des deux branches"

**Tag**

Une dernière notion importante à connaître lorsque l'on utilise git est le concept de tag. Les tags sont des références vers un point particulier de l'historique des modifications.

Un tag représente donc un commit en particulier, alors qu'une branche représente une succession de commits. Les tags sont notamment utilisés pour indiquer un point important dans l'historique des modifications, par exemple des commits correspondants à des versions déployées de l'application (c'est à dire version x.y.z, par exemple 1.0.1).

Pour créer un tag, il suffit d'exécuter la commande git tag suivi du nom du tag :

Créez un tag 1.0.0

git tag 1.0.0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Qu'est-ce que GitHub ?

GitHub est une plateforme en ligne de développement collaboratif lancée le 10 avril 2008. Exploitant le logiciel de gestion de versions décentralisé Git, la plateforme permet à tout utilisateur d'y héberger ses projets et de les rendre accessibles au développement coopératif grâce à un système de dépôts, en anglais, repository, retraçant les avancées et modifications apportées aux projets. Outre ses facettes techniques qui attirent les développeurs, l'un des aspects majeurs de GitHub est son caractère social prononcé.

Avec plus de 50 millions d'utilisateurs inscrits, le site met l'accent sur les interactions entre ceux-ci en proposant des fonctionnalités de suivi d'utilisateurs, suivi de projets, de discussions et bien d'autres encore, le tout dans une optique de laisser la vaste communauté s'approprier la plateforme.  
Dès lors qu'une entreprise ou une équipe privée de développeurs cherche à intégrer GitHub à leur mode de fonctionnement, la version gratuite n'est plus suffisante et les organisations sont poussées vers des forfaits payants.

### **Le marché de la gestion de code**

Naturellement, GitHub n'a pas le monopole absolu du développement collaboratif en ligne. Également construit autour de Git, GitLab est sans doute la meilleure alternative à GitHub. Open source, l'outil est privilégié par les entreprises pour son approche DevOps et ses outils d'administration organisationnelle. À d'autres niveaux, on peut également citer les plateformes de gestion de code proposées par Google et Amazon : Cloud Source et AWS CodeCommit, pouvant également être de bonnes alternatives si l'organisation choisie se base déjà autour des services cloud de Google ou AWS.

### **Se lancer sur GitHub**

GitHub Desktop est un outil visant à simplifier les interactions avec GitHub en permettant aux utilisateurs de s'affranchir des lignes de commandes Git tout en gardant une vision d'ensemble claire de l'état des projets administrés. Pour les moins férus de shell c'est l'alternative idéale, en revanche d'autres plus expérimentés diront que l'application limite les possibilités de part son interface trop simplifiée.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Premiers pas sur la plateforme

### **Les dépôts / repositories**

La collaboration sur GitHub s'effectue principalement en termes de dépôts ou repositories en anglais, repo pour faire court. Si l'on souhaite commencer un projet et le partager, on peut créer notre propre dépôt, ou bien participer à un projet en cours sur un dépôt déjà existant. Dans les deux cas, il faut faire le lien entre un dépôt distant hébergé sur GitHub, et un dépôt local hébergé sur votre machine.

Pour créer un nouveau dépôt, il suffit de cliquer sur le bouton New accessible sur la barre latérale gauche de la page d'accueil, puis de le nommer et de le paramétrer à votre guise.

Depuis la page d'accueil, créez un nouveau dépôt privé dont vous choisirez le nom. Ajoutez-y un fichier README.md qui servira de description pour votre dépôt.

Votre dépôt devrait maintenant apparaître dans la barre latérale gauche. Accédez à celui-ci en cliquant dessus.

Add / Commit / Push

Dans votre dépôt local, créez un nouveau fichier fichier\_1.txt dans lequel vous écrirez une phrase de votre choix telle que 'Ceci est mon premier fichier'.

Nous avons donc ajouté un fichier à notre dépôt local, mais il n'est pas encore apparu sur notre dépôt distant. Voyons l'état d'avancement de notre dépôt à l'aide de git status.

Notre fichier est dit "untracked" et apparaît en rouge dans le message, il faut donc l'ajouter avec git add

Ajoutez le fichier fichier\_1.txt à la liste des fichiers à prendre en compte dans le prochain commit, puis afficher à nouveau le statut de notre dépôt.

# Ajout du fichier

git add fichier\_1.txt

# Statut du dépôt

git status

Cette fois-ci, notre fichier n'est plus "untracked", nous pouvons donc passer au commit pour enregistrer la modification effectuée.

Pour ajouter un message, il suffit d'ajouter l'argument -m à la fonction git commit. Par exemple :

git commit -m 'Ajout de fichier\_1.txt'

git status

La modification a bien été enregistrée, il nous suffit simplement de les "pousser" vers le dépôt distant.

Tant que l'on ne pousse pas nos modifications sur le serveur distant, il est toujours possible d'annuler ou de revenir à un commit antérieur. Pour retracer l'historique de nos commits sur la branche principale, nous pouvons utiliser la commande git log, elle retournera l'ensemble des commits, accompagnés de leurs identifiants et des messages associés.

Pour communiquer nos modifications au dépôt distant, la commande à effectuer est git push comme mentionnée dans le message obtenue après le git status.

Pousser les modifications faites sur le dépôt local vers votre dépôt distant.

Vérifiez que les modifications ont bien été appliquées en vous rendant sur la page web GitHub de votre dépôt.

# Sur le dossier du dépôt

git push

En actualisant la page web GitHub de notre dépôt, le fichier fichier\_1.txt a bien été ajouté.

### **Fetch / Pull**

Dans le cadre d'un projet collaboratif, il faut s'attendre à ce que d'autres participants effectuent des modifications et les incorporent au dépôt distant. Dès lors, le dépôt local sur lequel on travaille est voué à se désynchroniser par rapport aux modifications faites par d'autres. Pour récupérer ces changements, on utilisera les commandes git fetch et git pull.

À l'instar de la partie précédente, récupérer des changements effectués sur le dépôt distant se fait en deux étapes :

1. Vérifier s'il y a eu des changements apportés au dépôt distant à l'aide de la commande git fetch.

Celle-ci téléchargera les métadonnées du dépôt sans pour autant télécharger les modifications faites. Elle permet donc de vérifier si des changements peuvent être importés sur le dépôt local.

1. Importer les changements effectués sur le dépôt distant vers le dépôt local à l'aide de la commande git pull.

### **Les branches**

Avec ce que l'on vient de voir, on se rend rapidement compte que pour ne pas avoir de conflits de versions entre les différents participants d'un projet, il est préférable que chacun puisse travailler sur sa propre branche avant de penser à modifier la branche principale.

Pour créer une branche, on peut le faire de deux façons différentes :

Depuis le site de GitHub, sur la page "< > Code" du dépôt.

Il suffit de cliquer sur le bouton déroulant associés aux branches dans la partie supérieure gauche du dépôt.

Par défaut vous serez sur la branche main (ou master). main étant le nouveau nom de la branche par défaut, master est l'ancien nom.

Une fois que cette étape est faite, pour synchroniser votre dépôt local avec le dépôt distant, il suffit de télécharger les métadonnées associées avec git fetch, puis pour se placer sur cette nouvelle branche sur le dépôt local, utilisez la commande suivante :

git checkout [nom-de-la-branche]

* Depuis le terminal en local

La méthode conseillée, car il est souvent préférable de partir du local, vers le distant plutôt que l'inverse. Pour instancier une nouvelle branche, assurons-nous avant tout que notre dépôt local est synchronisé avec notre dépôt distant à l'aide des commandes git fetch et git pull. Puis dans un second temps, on effectuera les commandes suivantes :

# Pour créer la branche

git branch [nom-de-la-branche]

# Pour se placer sur celle-ci

git checkout [nom-de-la-branche]

Ensuite, il faut pousser ces informations sur le dépôt distant à l'aide de la commande suivante :

git push -u origin [nom-de-la-branche]

Où -u est l'argument --set-upstream permettant à notre branche d'être traçable, origin correspond par défaut au nom de notre dépôt sur lequel on pousse nos modifications.

# Création de la nouvelle branche

git branch une\_branche

# On se place sur la nouvelle branche

git checkout une\_branche

# Une fois fait on pousse cette nouvelle branche sur notre dépôt distant

git push -u origin une\_branche

# Notre branche faite, on peut créer notre nouveau fichier texte

nano fichier\_2.txt

# On le remplit avec du texte au choix, puis on l'ajoute au prochain commit

git add fichier\_2.txt

# On commit puis on push

git commit -m "ajout de fichier\_2.txt sur une nouvelle branche"

# Toujours sur la branche

git push

Nous venons donc de créer une nouvelle branche et d'y ajouter un nouveau fichier. Lorsque l'on regarde sur la page du dépôt distant, ce fichier existe dans notre branche de développement mais n'existe pas sur notre branche principale main.

Supposons que nous ayons fait les ajouts nécessaires sur notre branche et que nous voulons maintenant partager notre travail avec les autres collaborateurs, nous devons faire basculer les modifications apportées à notre branche sur la branche principale du dépôt. Pour cela, on fusionnera (merge en anglais) notre branche de développement avec notre branche principale.

Avant d'effectuer la fusion il peut être intéressant de regarder les différences entre les branches, pour cela on peut utiliser la commande git diff qui s'utilise comme suit :

git diff [nom-de-la-première-branche] [nom-de-la-seconde-branche]

Cette commande listera tous les fichiers, dossiers et autres modifications qui différent entre les deux branches. Une fois que l'on s'est assuré que ces modifications sont les bonnes, on peut passer à la fusion des branches à l'aide de la fonction git merge depuis la branche principale qui sera à la réception de cette fusion. La fonction s'utilise comme suit :

git merge [nom-de-la-branche-à-merge]

# On se place sur notre branche principale

git checkout main

# On fusionne les deux branches

git merge une\_branche

Une fois fait, il suffit de pousser les modifications sur le dépôt distant à l'aide de git push et notre branche principale sera à jour. La branche de développement est maintenant obsolète, on peut la supprimer en local à l'aide de la commande suivante:

git branch -D [nom-de-la-branche]

Pour la supprimer sur le dépôt distant, il faudra aller sur le site de votre dépôt, cliquer sur branch et supprimer la branche devenue obsolète.

### **Les Pull Requests**

Pour encadrer ses fusions de branches, GitHub propose un système de Pull Request (PR). Dans la plupart des cas, une personne de l'organisation sera désignée pour surveiller et autoriser les fusions de branches vers la branche principale du projet. Cet outil a pour but de limiter les merge conflicts ou conflits de modifications, qui peuvent se produire lorsque deux acteurs d'un projet modifient en parallèle sur deux branches différentes, des parties communes du projet. Dans ce genre de situation, les acteurs soumettront une Pull Request à l'administrateur du projet et ce dernier devra se charger d'effectuer les modifications nécessaires pour résoudre les conflits afin d'autoriser le merge des branches.

# Création de la nouvelle branche

git branch une\_nouvelle\_branche

# On se place sur la nouvelle branche

git checkout une\_nouvelle\_\_branche

# Une fois fait on pousse cette nouvelle branche sur notre dépôt distant

git push -u origin une\_nouvelle\_branche

# Notre branche faite, on peut créer notre nouveau fichier texte

nano fichier\_3.txt

# On le remplit avec du texte au choix, puis on l'ajoute au prochain commit

git add fichier\_3.txt

# On commit puis on push

git commit -m "ajout de fichier\_3.txt sur une nouvelle branche"

# Toujours sur la branche

git push

Une fois cette étape faite, rendez vous sur la page principale de votre dépôt et allez sur l'onglet Pull Request. Soumettez une nouvelle pull request avec le bouton "New pull request", décrivez le contenu de vos changements apportés dans l'espace prévu à cet effet. Vous pouvez également assigner un relecteur (reviewer), et un label pour décrire davantage vos changements. Du côté administrateur, ce dernier peut voir quels fichiers ont été ajoutés, modifiés ou supprimés, apporter ses propres changements à ces mêmes fichiers et finalement, autoriser le merge de la branche. GitHub notifie automatiquement l'administrateur de la possibilité de fusionner les branches. Le cas contraire, il sera également notifié d'un merge conflict qu'il faudra résoudre en modifiant les fichiers qui posent problème.

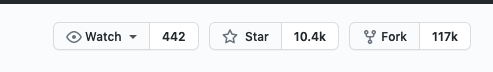
Enfin, si aucun problème n'est détecté par GitHub et l'administrateur, ce dernier peut autoriser la pull request et un nouveau commit sera créé comme précédemment dans le paragraphe introduisant le merge. GitHub proposera de supprimer la branche désormais obsolète, en local il nous suffira de pull les changements apportés sur la branche principale, puis de supprimer la branche locale elle aussi devenue obsolète.

### **Fork**

Une autre manière de participer à un projet est de fork ce dernier. En résumé, il s'agit de copier l'intégralité d'un dépôt GitHub créé par quelqu'un d'autre et d'en faire un de vos dépôts personnels, le cloner, le modifier... comme si c'était le votre. Ainsi, vous pourrez travailler dessus sans modifier l'architecture propre du dépôt, essayer vos changements de votre côté et une fois que ceux-ci vous conviennent, et que vous pensez qu'ils contribueront au projet originel, vous pouvez soumettre une pull request.

C'est notamment grâce à ce système que les projets open source trouvent des contributeurs : n'importe qui peut forker le dépôt public d'un projet open source, tenter de l'améliorer, et proposer ces modifications via une pull request. Ces changements seront alors examinés par les administrateurs du projet open source, puis acceptés ou refusés. L'administrateur peut également discuter avec le contributeur via la pull request, notamment pour lui demander d'améliorer ses modifications avant qu'il les intègre (mise à jour de la documentation, ajout de test, de commentaires, etc.).

Pour "Forker" un dépôt, cliquez sur le bouton éponyme situé en haut à droite du dépôt.



Dès lors, comme si c'était votre dépôt, vous pourrez le cloner et effectuer toutes les actions qui vous semblent positives au développement du projet.

### **Issues**

Vous pouvez également créer des "issues" (problèmes) sur un de vos dépôt, ou un dépôt public d'un projet open source. Une issue peut servir à faire remonter un bug, demander l'ajout d'une fonctionnalité, ou simplement poser une question quant au projet.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Workflows et Process

### **Git Flow**

Git Flow repose sur un branching model (modèle de branche) relativement complexe. Il comporte les branches suivantes :

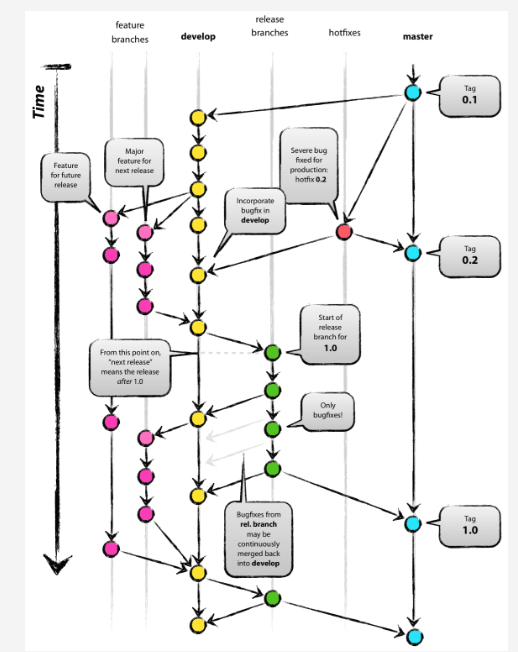
main (ou master) : C'est la branche stable, dont les différents commits reflètent les versions du code qui ont été déployées en production. Pour chaque commit de cette branche, on crée en général un tag qui porte le nom de la version (par exemple 1.0.1)

develop : C'est la branche de développement, à partir de laquelle les développeurs vont créer les feature branches

feature branches : ce sont les branches créées à partir de develop pour travailler sur une nouvelle fonctionnalité (feature). Par exemple, si vous êtes chargé d'implémenter l'authentification sur votre application, vous devrez créer une branche feature/auth à partir de develop. Une fois l'authentification implémentée, vous fusionnerez vos changement dans la branche develop (via une pull request).

release branches : Une fois que les features que l'on souhaite déployer dans la prochaine release (sortie de la prochaine version de l'application) ont été développées et fusionnées dans develop, on peut créer une branche de release à partir de develop, dont le nom contient en général le numéro de la version (par exemple release/1.0.1). Ce code sera ensuite testé extensivement, et on pourra effectuer quelques corrections de bugs (bug fix) si nécessaire. Par la suite, on fusionnera la branche de release à la fois dans la branche main / master et dans la branche develop, et on pourra déployer la nouvelle version en production.

hotfixes : ces branches sont créées à partir de main / master afin de corriger les bugs importants en production. Ces branches sont par la suite fusionnées à la fois dans la branche main / master et dans la branche develop.

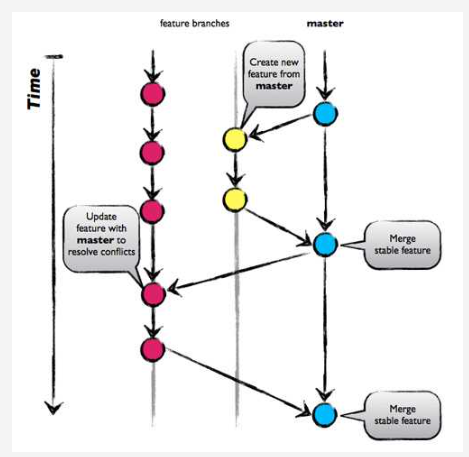


### **GitHub Flow**

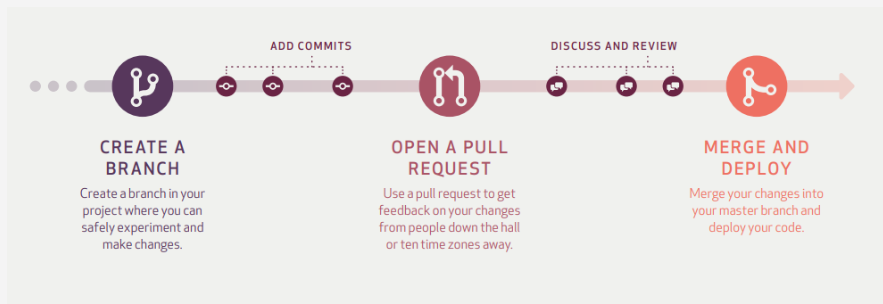
Comme nous l'avons vu, Git Flow est un workflow avec un branching model relativement complexe. Si ce modèle a des avantages, il peut néanmoins s'avérer lourd dans certains cas, notamment lorsque les releases sont très fréquentes (par exemple hebdomadaire voire quotidienne).

Certains workflows ont été défini dans le but de proposer un modèle plus simple et plus adapté dans le cas où l'on souhaite déployer les modifications dès qu'elles ont été développées, sans les regrouper au sein d'une release. C'est notamment le cas de GitHub Flow.

Le branching model est donc beaucoup plus simple : il comporte une branche main (ou master) qui contient les versions "déployables" de l'application. A partir de cette branche, les développeurs créeront de nouvelles branches pour travailler sur des features ou des bug fix. Après avoir implémenté la nouvelle feature ou corrigé le bug, la branche sera directement "mergée" dans main / master via une Pull Request :



Comme les changements sont intégrés directement dans main / master, il faudrait bien faire attention à prendre le temps de valider ces changements dans la Pull Request : celle-ci permet en effet aux différents contributeurs du projet de discuter des changements apportés. En particulier, un administrateur du projet devra vérifier que ces changements sont conformes aux attentes, qu'ils sont bien documentés et testés, et qu'ils n'introduisent pas de bugs.



Process complet

Les deux workflows vus précédemment définissent des branching models qu'une équipe de développement pourra adopter, en fonction de ses contraintes et de son organisation. Mais d'autres aspects devront être couverts pour définir le process complet :

Le process de review des Pull requests : On a insisté sur l'importance de valider les changements proposés via les pull requests. Sur GitHub (ainsi que sur de nombreuses plateformes similaires), on peut notamment imposer qu'un certain nombre de reviewers (relecteurs) valident les changements avant de pouvoir la fusionner. Ceci peut être utile dans certaines équipes, afin de s'assurer de la qualité des modifications apportées. Pour d'autres, notamment des petites équipes, imposer plusieurs reviewers peut s'avérer lourd. Encore une fois, chaque équipe doit mettre en place les process qui lui conviennent.

Mise en place de test : On peut (et même on doit) écrire des tests pour l'application développée, les automatiser, et par exemple imposer que les tests se passent avec succès pour pouvoir fusionner une Pull Request.

Les conditions de déploiement : il faudra que l'équipe décide quand et comment les nouvelles versions devront être déployées. On peut notamment automatiser les déploiements à partir d'évènements sur un dépôt (création d'un tag, merge d'une PR) : c'est ce que l'on appelle le Continuous Delivery (Livraison continue).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Introduction au CI / CD

### **A) Contexte**

Dans un projet typique d'apprentissage automatique (Machine Learning ou ML), la solution finale proposée doit fournir une formation et une implémentation automatisées des modèles choisis. Mais qu'en est-il de l'intégration et du déploiement des modèles précédemment évoqués ? C'est là qu'intervient la méthode CI/CD : Cette solution d'intégration continue / de déploiement continu fournit un pipeline de bout en bout qui parachève le cycle d'un projet global et garantit les performances du modèle.

Initialement, l'intégration et le déploiement continus constituent une technique DevOps permettant de mettre en place un pipeline automatisé à des fins de production en :

* rationalisant le code
* testant l'application
* déploiement / mise en production

Le DevOps désigne un ensemble de processus visant à raccourcir le cycle de développement d'un système en assurant la livraison continue de logiciels de haute qualité.

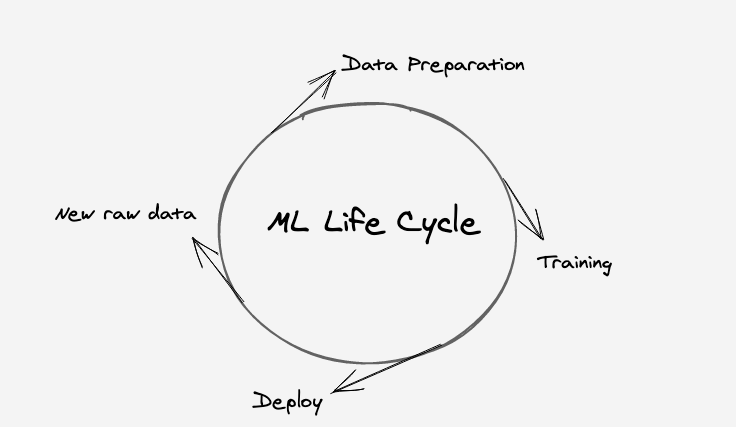
Le MLOps, quant à lui, est le processus d'automatisation et d'industrialisation des applications et des flux de travail d'apprentissage automatique. Le CI/CD représente ici un flux de travail d'automatisation du pipeline d'apprentissage automatique à travers les opérations suivantes :

* construction du modèle
* test
* déploiement

Cela évite également au data scientist de s'occuper ou de se préoccuper de ce processus, en éliminant toute négligence humaine et en améliorant continuellement l'efficacité du modèle grâce à une surveillance permanente du modèle d'apprentissage automatique. Ainsi, toute modification de la structure du modèle est facilitée et son développement est automatisé pour une livraison fiable.

Comme le flux de travail du CI/CD va automatiser les différentes étapes d'un projet d'apprentissage automatique, faisons un petit rappel du cycle de vie typique d'un projet d'apprentissage automatique.

### **B) Cycle de vie du projet**



* Préparation des données : Les données sont généralement présentées initialement sous forme brute. Il est donc nécessaire d'effectuer quelques étapes de prétraitement de ces ensembles de données afin de les adapter à l'étape de la modélisation. Ce traitement est généralement effectué par le Data Scientist (scientifique des données) ou parfois par le Data Analyst (l'analyste des données) et peut nécessiter l'utilisation d'outils tels que Apache Spark, MySQL ou Python, et de bibliothèques telles que Pandas ou Numpy.
* Entraînement du modèle : Cette étape menée par le Data Scientist est le point central du cycle de vie du projet : l'objectif de la formation du modèle est de répondre à un problème spécifique en concevant et en paramétrant l'algorithme approprié. Cette itération nécessite généralement l'importation d'outils tels que les frameworks TensorFlow, PyTorch, ou la bibliothèque Scikit-Learn.
* Déploiement du modèle : Une fois le modèle prêt, Machine Learning Engineer (l'ingénieur apprentissage automatique) ou Data Engineer (l'ingénieur des données) se charge de le mettre à la disposition du client pour une utilisation facile et pratique.
* Nouvelles données brutes : Bien que l'on puisse s'attendre à ce que le projet touche à sa fin, il arrive très souvent que le Data Engineer reçoive de nouvelles données brutes disponibles après ces étapes. Elles doivent donc être intégrées dans le cycle décrit ci-dessus pour affiner et améliorer les performances du modèle développé précédemment.

# **B - Les concepts CI/CD**

1. CI : Continuous Integration (intégration continue)

CI est l'abréviation de Continuous Integration. Cette pratique regroupe les concepts d'ingénierie logicielle autour de l'automatisation de la construction et des tests du modèle, du contrôle de version et du release. Par release, nous entendons ici explicitement l'action de charger le code dans le dépôt GitHub correspondant.

1. CD : Continuous Delivery/Deployment (déploiement continu)

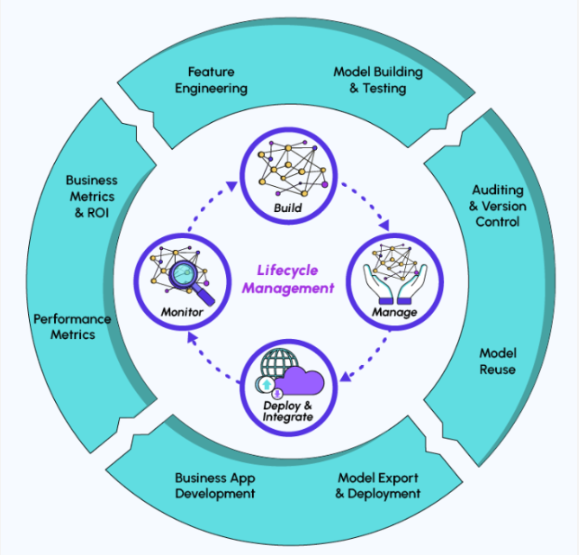
CD est l'abréviation de Continuous Delivery ou Continuous deployment. Ce concept représente le déploiement continu du modèle, le formatage et la mise en place d'un environnement de type production pour permettre une intégration automatisée. Concernant l'application, l'étape CD comprend le test, le déploiement et la configuration automatisée de l'application.

1. CI/CD avec un problème d'apprentissage automatique (ML)

L'intégration et le déploiement continus (CI/CD) sont depuis longtemps une pratique courante pour la majorité des systèmes logiciels en offrant une formation et une application continues et automatiques des modèles ML, les systèmes d'apprentissage automatique peuvent faire pareil.

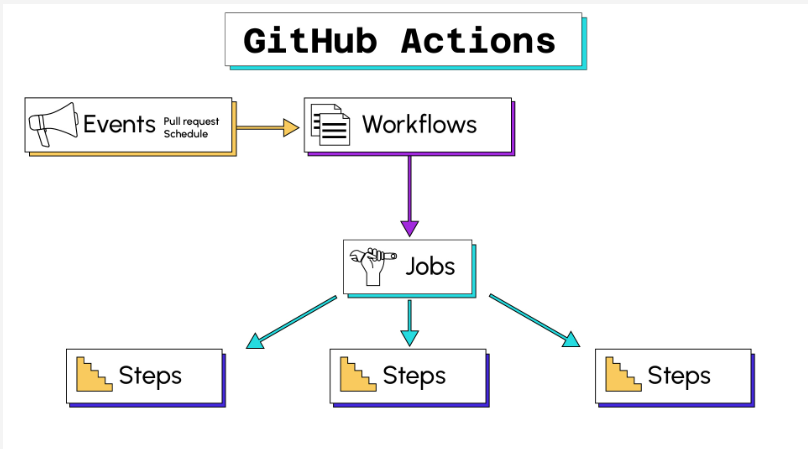
Les applications d'apprentissage automatique qui utilisent le CI-CD offrent un canal complet qui ferme la boucle de rétroaction à tous les niveaux et maintient les performances des modèles d'apprentissage automatique. Elles peuvent également combler le fossé entre les processus scientifiques et d'ingénierie en supprimant les obstacles entre les données, la modélisation, le traitement et le retour.

Ci-dessous, vous pouvez voir la gestion du cycle de vie typique d'un projet d'apprentissage automatique et les étapes CI/CD associées.



Une remarque toutefois concernant la différence entre les configurations de livraison continue et de déploiement dans un cycle de vie : Comme pour les deux configurations l'intégration continue comprend la construction et le test du modèle, l'étape de déploiement du modèle en production est un déploiement continu automatisé alors qu'il est manuel dans la configuration de livraison continue.

## C - Outils : Actions Github



Que sont les GitHub actions ?

GitHub actions est un outil qui vous permet d'automatiser vos flux de travail de développement de logiciels. GitHub actions est l'outil le plus utilisé pour construire des pipelines CI/CD.

Disons que vous avez un référentiel GitHub et que, sur ce référentiel, un événement peut se produire ; par exemple, quelqu'un a chargé un script, une demande d'extraction a été ouverte, un problème a été créé ou fermé, etc.

Et ces choses sont appelées événements. Et nous aimerions définir un flux de travail (ou workflow) lorsque ces événements se produisent. Un workflow peut contenir un ou plusieurs jobs (tâches) et chaque job s'exécute sur sa propre machine virtuelle.

Une machine virtuelle (VM) sera démarrée pour chaque job que nous avons dans notre workflow et cette VM peut être une machine Windows, Linux ou Mac OS. Nous pouvons définir le système d'exploitation directement dans le flux de travail. Pour chaque job, nous définissons un ensemble d'étapes et chaque étape doit effectuer une action. L'étape qui exécute ces jobs est appelée un Github runner.

En bref, chaque flux de travail se compose de plusieurs concepts fondamentaux différents. Ceux-ci comprennent :

* Les événements : Les événements sont des déclencheurs définis qui lancent un flux de travail. Ils peuvent être configurés pour s'exécuter sur des branches de codage spécifiques dans un dépôt donné sur GitHub.
* Jobs : Les tâches sont un ensemble d'étapes qui s'exécutent sur le même exécutant. Chaque tâche s'exécute dans sa propre VM et en parallèle avec les autres tâches.
* Steps : Les steps sont des tâches individuelles qui exécutent les commandes d'un job. Il peut s'agir d'une action ou d'une commande shell. Toutes les étapes d'un job s'exécutent sur le même runner.
* Actions : Une action est une commande qui est exécutée sur un runner et l'élément central de GitHub Actions, qui porte son nom.
* Runners : Un runner (coureur) est un serveur GitHub Actions. Il est à l'écoute des tâches disponibles, les exécute en parallèle et signale la progression, les journaux et les résultats. Chaque runner peut être hébergé par GitHub ou auto-hébergé sur un serveur localisé. Les runners hébergés par GitHub sont basés sur Ubuntu Linux, Windows et macOS.

Notre premier flux de travail

Nous allons exécuter un flux de travail décomposé en 3 tâches sur différentes versions d'Ubuntu.

1 - Créez un dépôt Github public dans votre compte Github nommé github\_actions.

2 - Clonez votre dépôt sur la VM Datascientest

git clone <https://github.com/your-username/github-actions.git>

3 - Créez un dossier nommé .github/workflows

cd github-actions

mkdir -p .github/workflows  
#pour un créer un ens. de dossiers et sous dossier.

cd .github/workflows

4 - Créez un fichier de flux de travail appelé my\_first\_workflow.yaml

nano my\_first\_workflow.yml

ame: Shell commands

on: [push]

jobs:

run-shell-command:

runs-on: ${{ matrix.os }}

strategy:

matrix:

os: [ubuntu-latest, ubuntu-20.04, ubuntu-18.04]

steps:

- name: echo a string

run: echo "Hello world"

- name: multiline script

run: |

python3 -V

touch my\_file\_with\_workflow.py

ls -l

Ce flux de travail comporte 2 étapes en 3 tâches (ou jobs)

Étape 1 :

Affichez un "hello world" dans la console.

Étape 2 :

Affichez la version de Python, la version disponible par défaut dans Ubuntu.

Créez un fichier nommé my\_file\_with\_workflow.py.

Listez les fichiers dans le dossier actuel

5 - Faites un commit (validation) et chargez votre dépôt, écrivez my first action en commentaire

git commit -m “my first action”

git push

Afin d'accéder à votre VM avec votre compte github, vous pouvez créer un jeton d'accès au lieu d'un mot de passe. Le mode de connexion par mot de passe n'est plus pris en compte.

Veuillez suivre les étapes suivantes à ce stade : créer une jeton d'accès

6 - Allez sur votre compte Github dans l'onglet Actions puis étapes :

7 - Une fois dans l'onglet action, cliquez sur votre flux de travail.

8 - Une fois le flux de travail terminé, téléchargez les journaux et voyez ce qui se passe.

Vous pouvez activer les notifications par mail ou push si une exécution réussit ou échoue :

Allez dans la section settings (paramètres) de votre compte.

Allez dans la section notification et trouvez la partie Actions.

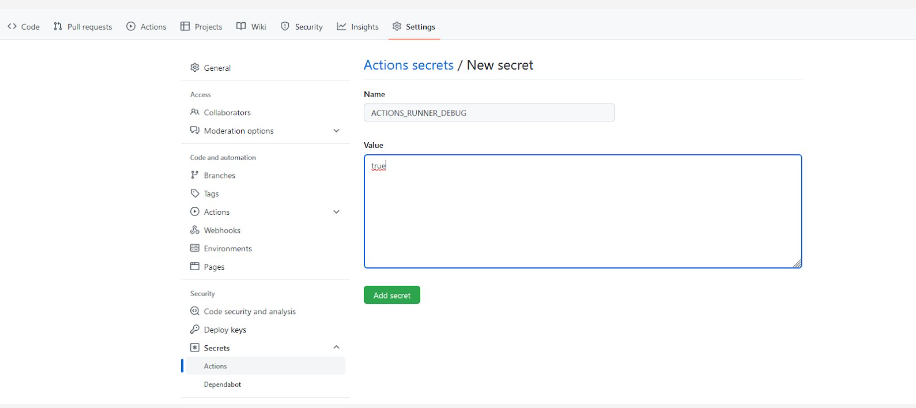
Vous verrez Notifications for workflow runs on repositories set up with GitHub Actions (notifications pour les exécutions de workflow sur les dépôts configurés avec GitHub Actions).

Cochez la case Send notifications for failed workflows only (envoyer des notifications uniquement pour les workflows qui ont échoué).

Dans un environnement de test, vous pouvez avoir besoin d'un mode de débogage pour voir comment fonctionne votre workflow et le modifier en conséquence. Vous pouvez activer le debug mode (mode débogage) dans votre dépôt.

9 - Dans le dépôt sur Github, cliquez sur " Settings ". Une fois dans l'onglet settings, cliquez sur "new repositories secret".

10 - Ajoutez la configuration suivante dans la fiche :



* NAME : ACTIONS\_RUNNER\_DEBUG
* KEY: true

Cliquez sur "Add secret"

Faites de même avec :

* NAME : ACTIONS\_STEP\_DEBUG
* KEY: true

Ensuite, pour tester la configuration, modifiez votre fichier de workflow my\_first\_workflow.yaml dans votre VM

Remplacez :

run: echo "Hello world"

Par :

run: echoo "Hello world"

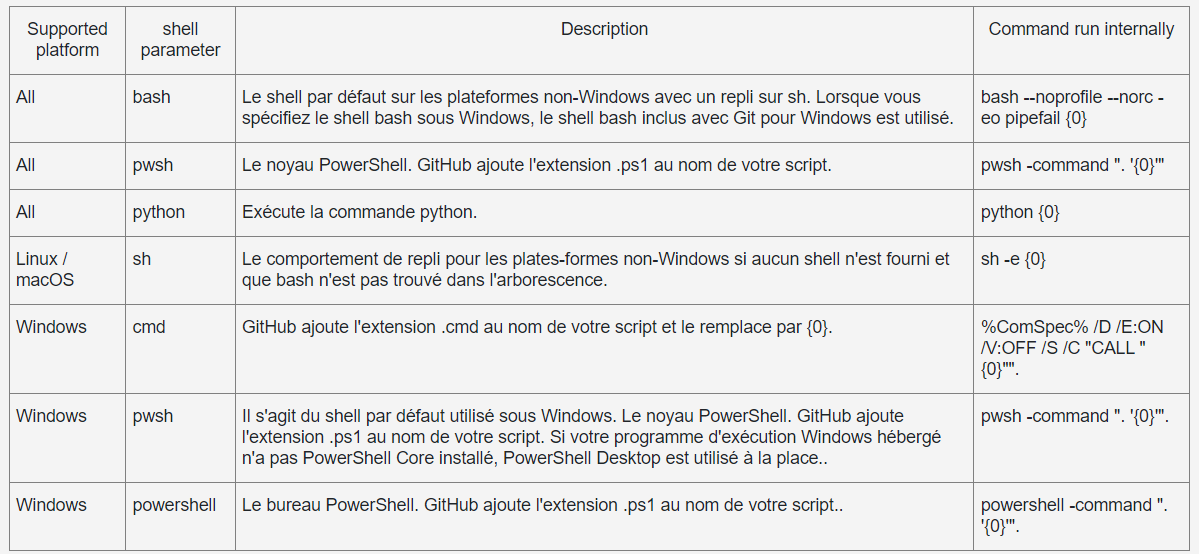
Cette ligne générera une erreur. Vous verrez le débogage dans l'onglet actions de github lors de l'execution du worflow.

Ne pas oublier de faire un commit et de push sur Github.

11 - Téléchargez les journaux de logs pour voir où l'erreur a été soulevée.

Vous venez de créer votre premier flux de travail et vos premières tâches. Et dans ces étapes, nous avons également exécuté quelques commandes. Vous pouvez aussi changer le shell que vous souhaiter utiliser.

Vous trouverez ici un tableau de tous les shells pris en charge dans les actions GitHub, ainsi que les plateformes compatibles :



Il y a donc le bash qui est l'interpréteur par défaut dans les machines virtuelles Linux et Mac OS et qui est compatible avec toutes les plateformes. Il peut également être utilisé sous Windows, cependant sous Windows l'option par défaut est le powershell.

Le Powershell ne fonctionne que sur Windows, nous ne pouvons donc pas l'utiliser sur un Mac ni sur une VM Linux. Cependant, vous pouvez utiliser le bash sur une VM Windows. En plus de cela, vous avez également d'autres shells comme le pwsh et le python.

Essayons à présent d'utiliser un autre shell dans notre tâche Ubuntu et construisons un exemple plus complexe.

1 - Créez un nouveau workflow dans votre machine appelé multiple\_shells.yaml et collez-y les lignes suivantes :

name: Multiple shells commands

on:

push:

branches:

- "main"

pull\_request:

types: [closed, assigned, opened, reopened]

jobs:

run-shell-command:

runs-on: ubuntu-latest

steps:

- name: python commands

run: |

import platform

print(platform.processor())

shell: python

run-windows-command:

runs-on: windows-latest

needs: ["run-shell-command"]

steps:

- name: current directory with powershell

run: Get-Location

- name: current directory with bash

run: pwd

shell: bash

- name: print some default env variables

run: |

echo "HOME: ${HOME}"

echo "GITHUB\_WORKFLOW: ${GITHUB\_WORKFLOW}"

echo "GITHUB\_ACTION: ${GITHUB\_ACTION}"

echo "GITHUB\_ACTIONS: ${GITHUB\_ACTIONS}"

echo "GITHUB\_ACTOR: ${GITHUB\_ACTOR}"

echo "GITHUB\_REPOSITORY: ${GITHUB\_REPOSITORY}"

echo "GITHUB\_EVENT\_NAME: ${GITHUB\_EVENT\_NAME}"

echo "GITHUB\_WORKSPACE: ${GITHUB\_WORKSPACE}"

shell: bash

2 - Faites un commit et chargez le nouveau fichier dans Github.

3 - Vous devriez voir 2 tâches en cours d'exécution dans l'onglet Action.

4 - Cliquez sur le flux de travail run-windows-command et parcourez les différentes étapes. Vous verrez toutes les variables imprimées dans l'étape print some default env variables.

Une image contenant texte

Description générée automatiquement

**Quelques explications:**

* La clé needs est utile dans le cas où vous voulez exécuter des tâches de manière "séquentielle". La tâche suivante ne sera exécutée que lorsque la précédente se terminera avec succès.
* Vous pouvez **déclencher** un événement et choisir sur quelle **branche** votre workflow doit s'exécuter. Ici nous définissons "Si j'applique un commit dans la branche master, lancer le workflow". Nous voulons également exécuter notre workflow en fonction du statut d'une **pull request**. Par exemple, "si je demande une pull request et que je fusionne la branche, teste le code avant de le fusionner dans la branche principale" Consultez la [documentation](https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows) pour voir tous les **événements** qui déclenchent un workflow.

Dans les actions Github, vous avez des **variables d'environnement** par défaut que vous pouvez utiliser pour effectuer des tâches :

* HOME qui est notre répertoire personnel dans notre machine virtuelle. Il est créé par les actions Gitub et ne peut pas être modifié.
* GITHUB\_WORKFLOW qui est le nom du workflow.
* GITHUB\_ACTION qui est un identifiant unique pour l'étape en cours d'exécution. Ainsi, puisque vous n'avez qu'une seule étape en cours d'exécution dans ce job, elle s'appelle run.
* GITHUB\_ACTIONS qui sera toujours vrai si vous exécutez un workflow GitHub. Cela peut être utile si vous exécutez certains tests dans votre code et que vous avez besoin de vérifier si vous exécutez ces tests sur le CI dans les actions GitHub ou localement sur votre machine.
* GITHUB\_ACTOR qui est le nom d'utilisateur du compte GitHub et GITHUB\_REPOSITORY qui est le nom d'utilisateur / le nom du dépôt.
* GITHUB\_EVENT\_NAME qui est l'événement qui a déclenché l'exécution de ce workflow. Dans notre cas, il s'agissait d'un push, mais cela pourrait être une pull request par exemple.
* GITHUB\_WORKSPACE est le chemin qui contient les fichiers de votre dépôt.
* GITHUB\_SHA qui est l'identifiant du commit qui a déclenché l'exécution de ce workflow. Nous avons la branche qui a déclenché l'exécution du workflow. Dans ce cas, celui-ci est push vers le master.

Toutes ces variables sont les plus utilisées mais cette liste n'est pas exhaustive. Consultez la documentation pour connaître toutes les variables d'environnement par défaut disponibles pour les actions Github.

Vous pouvez également définir des variables d'environnement personnalisées pour authentifier et valider les modifications de vos dépôts dans le programme d'exécution. Vous verrez toutes les étapes dans le module suivant.

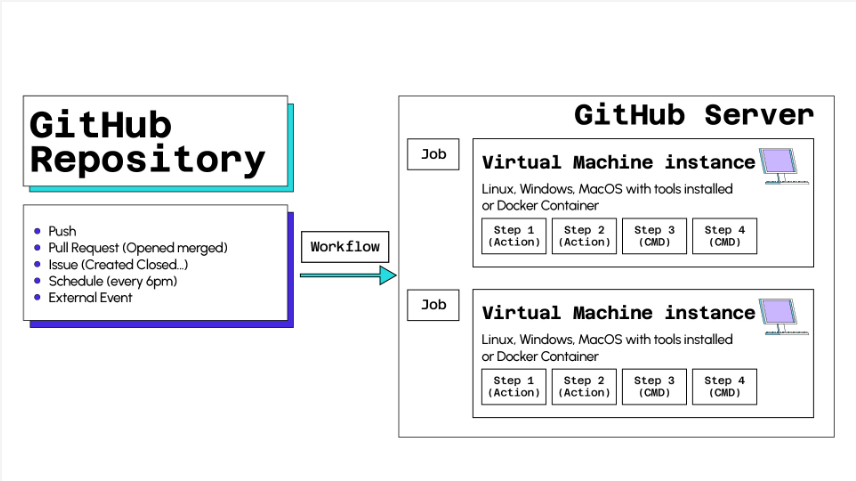
#### **Conclusion**

Les pipelines CI/CD sont généralement complexes et comportent de nombreux outils qui vont du test des applications à l'intégration en production. Vous pouvez simplifier le processus avec des intégrations Docker et vous permet de spécifier quelle version de Python vous voulez utiliser et ensuite connecter votre code à un environnement cible et une plateforme d'application.

Vous avez vu comment créer un flux de travail et quelques tâches à l'intérieur de ce flux de travail et aussi comment créer des étapes à l'intérieur de chaque tâche. Nous avons également vu comment exécuter des commandes shell dans une étape, ce qui peut être très utile, mais nous pouvons néanmoins utiliser des actions dans notre workflow pour faire beaucoup plus de choses.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Événements et actions Github étape par étape



Première étape

Tout d'abord, vous devez créer un script python qui contient des fonctions avec des assertions à tester. Pytest analysera toutes les fonctions de vos scripts mais il y a une condition : le nom de vos fonctions doit commencer par test\_.

Créons donc ce script nommé file\_test.py

import pytest

def test\_calc\_addition():

# Fonction test du résultat de 2+4

output = 2+4

assert output == 6

def test\_calc\_substraction():

# Fonction test du résultat de 2-4

output = 2-4

assert output == -2

def test\_calc\_multiply():

# Fonction test du résultat de 2\*4

output = 2\*4

assert output == 8

def test\_coucou():

# Fonction test si la résultat renvoie 'hello'

output='hello'

assert output == 'hello'

Construire notre action

Maintenant que nous avons notre fichier de tests, faisons notre workflow. Allez dans votre dossier contenant votre dépôt, créez un dossier secret nommé .github et un sous-dossier nommé workflows et créez votre fichier.

Créez control\_tests.yaml et attribuez un nom à notre flux de travail, définissez l'action sur "push"

name: Pytest & flake8 control Push

on: push

Définissez un id de tâche, nommez-le et définissons sur quelle instance nous voulons l'exécuter.

jobs:

qa:

name: Check tests

runs-on: ubuntu-20.04

Maintenant, configurons nos étapes. Nous avons 5 étapes :

* Un contrôle du dépôt
* Configuration de Python sur la VM
* L'installation des paquets
* L'exécution de nos tests
* Le contrôle de la convention Pep8

steps:

- name: Checkout on master

uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@master

with:

python-version: "3.x"

Dans la première étape, nous avons le mot clé uses qui fait appel à un workflow réutilisable déjà existant sur github. Ensuite nous installons Python sur l'instance avec un autre worflow qui existe déjà. et nous spécifions la version de Python avec le mot clé with.

Dans les étapes suivantes, nous allons installer deux paquets et exécuter pytest et flake8

- name: Install pytest and flake8

run: |

pip install pytest

pip install flake8

- name: Run tests

run: |

pytest

- name: Run flake8

uses: py-actions/flake8@v2

Avec le mot clé run, il suffit de lancer cette commande dans le shell Ubuntu sur l'instance. Pour utiliser flake8, vous devez utiliser un workflow déjà existant sur le marketplace github.

Voici votre workflow final :

name: Control Push

on: push

jobs:

qa:

name: Check tests

runs-on: ubuntu-20.04

steps:

- name: Checkout on master

uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@master

with:

python-version: "3.x"

- name: Install pytest and flake8

run: |

pip install pytest

pip install flake8

- name: Run tests

run: |

pytest

- name: Run flake8

uses: py-actions/flake8@v2

Maintenant, vous n'avez plus qu'à faire un commit et charger votre code dans votre dépôt.

Allez sur votre dépôt sur www.github.com, cliquez sur action et vous pourrez voir votre workflow s'exécuter.

Une image contenant texte

Description générée automatiquement

Maintenant, cliquez sur check tests (vérifier les tests) et vous verrez alors pourquoi notre action a échoué.

Une image contenant texte

Description générée automatiquement

Nous observons que les tests sont passés mais que l'étape flake8 a échoué. En examinant notre script, on constate que nous n'avons pas la convention de code Pep8. Vous pouvez corriger cela en effectuant un réusinage de votre code et en faisant attention aux messages d'erreur.

Dans cet exemple, à l'intérieur du script de workflow, vous pouvez remarquer que nous avons une ligne uses contenant actions/checkout@v3. Github Marketplace contient des outils qui ajoutent des fonctionnalités. Dans notre exemple, nous avons utilisé une action appelée checkout qui nous permet de recopier notre repository sur le serveur distant sur lequel tourne le workflow pour que le job puisse avoir accès à nos fichiers. Vous pouvez construire votre propre action et la publier sur le marketplace.

#### **Variable d'environnement**

Présentons un concept très utile dans l'action github appelé variable d'environnement. Une variable d'environnement est une variable dont la valeur est définie en dehors du programme et peut être utilisée pour déterminer des informations spécifiques à votre ordinateur. Vous pouvez définir une variable d'environnement pour l'ensemble du workflow en utilisant le niveau supérieur du fichier de workflow, seulement pour une tâche spécifique ou seulement pour une étape à l'intérieur d'une tâche.

Voici un bon exemple pour l'illustrer:

name: salutation sur un jour quelconque

on: workflow\_dispatch

env:

DAY\_OF\_WEEK: Lundi ## Variable d'environnement pour l'ensemble du flux de travail

jobs:

greeting\_job:

runs-on: ubuntu-latest

env:

Greeting: Salut ## Variable d'environnement pour la tâche

steps:

- name: "Dites Salut Mona nous sommes lundi"

run: echo "$Greeting $First\_Name. Nous sommes $DAY\_OF\_WEEK!"

env:

First\_Name: Mona ## Variable d'environnement uniquement pour l'étape "Dites Salut Mona nous sommes lundi" de la tâche "greeting\_job"

L'exemple ci-dessus montre trois variables d'environnement personnalisées utilisées dans une commande echo : $DAYOFWEEK, $Greeting, et $First\_Name.

Il existe un autre moyen de définir les variables d'environnement de manière plus sûre. Si vous ne voulez pas afficher certaines informations confidentielles, vous pouvez utiliser github secret pour ne pas afficher le code. Cela peut être très utile pour les identifiants de base de données ou les identifiants d'API par exemple. Pour créer des secrets, vous devez être le propriétaire du dépôt.

1- Sur GitHub.com, accédez à la page principale du dépôt.

2- Sous le nom de votre dépôt, cliquez sur Settings.

3- Bouton Repository settings (paramètres du dépôt).

4- Dans la barre latérale gauche, cliquez sur Environnements.

5- Cliquez sur l'environnement auquel vous voulez ajouter un secret.

6- Sous Secrets d'environnement, cliquez sur Ajouter un secret.

7- Tapez un nom pour votre secret dans la zone de saisie Name.

8- Saisissez la valeur de votre secret.

9- Cliquez sur Add secret.

Voici un autre exemple :

create\_commit:

runs-on: ubuntu-latest

permissions:

contents: write

steps:

- name: Push a file to our repository

run: |

pwd

ls -a

git init

git remote add origin "https://$GITHUB\_ACTOR:${{ secrets.GITHUB\_TOKEN }}@github.com/$GITHUB\_REPOSITORY.git"

git config --global user.email "my-bot@bot.com"

git config --global user.name "my-bot"

git fetch

git checkout main

git branch --set-upstream-to=origin/main

git pull

pytest >> log\_pytest.txt

ls -a

git add .

git commit -m "random file from action"

git push

Dans ce code, nous voulons charger un fichier dans notre dépôt distant github. Certaines opérations comme push ou pull requièrent des informations d'identification comme votre nom d'utilisateur ou votre mot de passe. Github a supprimé le mot de passe pour le remplacer par un jeton d'accès personnel pour une durée limitée ou pas. Le jeton peut être appelé et utilisé comme une variable d'environnement appelée dans notre script : GITHUB\_TOKEN. Dans cette action nous avons aussi 2 variables d'environnement nommées GITHUB\_ACTOR et GITHUB\_REPOSITORY. Cette action initialise un dépôt git à l'intérieur de la VM, définit les informations d'identification du compte, se connecte à votre dépôt distant, extrait le contenu puis exécute pytest et stocke les logs dans un fichier nommé log\_pytest.txt. Ensuite, nous validons ce fichier et nous le chargeons dans le dépôt.